Evolutionary behavior testing of commercial computer games

Ben Chan, Jörg Denzinger, Darryl Gates, Kevin Loose
Dept. of Computer Science
University of Calgary
Calgary, Canada
Email: {chanben, denzinge, gatesj, kjl}@cpsc.ucalgary.ca

John Buchanan
Electronic Arts Inc.
4330 Sanderson Way
Burnaby, Canada
Email: juancho@ea.com

***Abstract*— We present an approach to use evolutionary learning of behavior to improve testing of commercial computer games. After identifying unwanted results or behavior of the game, we propose to develop measures on how near a sequence of game states comes to the unwanted behavior and to use these measures within the fitness function of a GA working on action sequences. This allows to find action sequences that produce the unwanted behavior, if they exist. Our experimental evaluation of the method with the FIFA-99 game and scoring a goal as unwanted behavior shows that the method is able to find such action sequences, allowing for an easy reproduction of critical situations and improvements to the tested game.**

I. Introduction

Computer games have, from the beginning, been seen as an application that requires some artificial intelligence to offer to human players worthy computer opponents, resp. a challenging and entertaining gaming experience. But the field of AI, while definitely interested and seeing an interesting application, mainly concentrated on having AI programs play well-known "human" games, like checkers, backgammon or chess. While these games naturally required to represent them on computers, so that we can call them computer games, a lot of people interpret the term computer games differently, referring to commercial computer games that simulate more or less complex virtual realities in which the human player or players interact with the virtual reality and usually also computer controlled players (referred to as NPCs, non-player characters). And while most of these commercial computer games refer to the control of NPCs as their "AI", only very few of these AIs are based on any real concepts from the research area AI.

In the last few years, perhaps driven by the successes of AI programs playing "human" games (like [6] or [4]) and the realization that commercial computer games provide a very "AI friendly" testbed for AI concepts (as pointed out by Laird, see [5], among others) due to having a full world already represented in a computer understandable format, more and more researchers and game companies are investigating the usage of AI techniques, like A* search for path finding, or neural networks or evolutionary methods to learn behavior, for commercial computer games. But while the goal of many researchers is to develop AI controls that adapt to the human players (perhaps even consistently beating them), for game companies this goal cannot be a high priority: their games are supposed to keep the human players entertained, which requires to walk a very fine line between making it too easy to win and too difficult. As a consequence, many game companies are very reluctant to field games for which they cannot predict how the game (AI) will act and consequently match-up to the human players.

While this perception of the usage of AI hopefully will change in the future, we think that a first step for introducing learning concepts from AI into large parts of the commercial computer games industry is to use these concepts to help game designers and programmers to better understand and test the behavior of their games. This can be achieved by learning human player action sequences that lead to either unwanted game behavior or to game states that allow to test certain ideas and expectations of the designer. Since games are becoming larger and more complex, the problem of finding such unwanted (or desired) game states also becomes more complex and therefore harder. In the long term, the game industry increasingly will have to rely on tools that help in finding such game states, preferrably on tools that do so automatically. In this paper, we present how ideas from evolutionary learning of (cooperative) behavior (see [2]) can be used to test computer team sport games, more precisely the game FIFA-99, for some kinds of behavior that was not wanted this way by the designers, thus providing an important step towards the kind of testing tools the industry will need.

Our evolutionary test system evolved action sequences that lead with more than a certain high probability to scoring a goal when executed, thus allowing a human player knowing such a sequence to easily win the game, since the game allows for exploiting this weakness. Evolutionary learning is well suited to achieve this, since it simulates some kind of intuition (that is used by humans that want to find such *sweet spots*, as such sequences often are called) and is able to deal with the indeterminism that is used by many commercial computer games to offer different game playing experiences every time the game is played (while balancing this with requirements on the realism of the game). Our test system naturally has found a lot of such scoring sequences, many of which showed acceptable game behavior from the designer's point of view (i.e. realistic soccer behavior and difficult enough for a human

to repeat), but there were also scoring sequences that were considered unrealistic and therefore should be taken care of by changing the game AI.

II. An interaction behavior based view on commercial computer games

In this section, we will present a rather abstract view on how human players interact with commercial computer games. We will also address some of the problems that arise with regard to testing commercial computer games.

If we look at how human players interact with commercial computer games, then we can see the interactions as a stream of inputs provided by the player to the game and these inputs have some influence on the *state* of the game. In the case of several human players, several input streams are combined within the game. Obviously, time plays a certain role in the treatment of inputs, but essentially there are only two ways how time can be included. Either every input of the player(s) leads to a new game state, which means that input streams define their own time, or new states are generated every time unit, where the definition of time unit is provided by the game. In the latter case, we can model time on the input side by having a "no-input" input and then reading the next input from the stream every time unit. Then sequences of "no-input" inputs can model every possible delay by the player.

More precisely, the inputs of a game player can be seen as *action* commands (with "no-input" being the action of not doing anything) given by the player to the game out of a set Act of possible actions. The game itself is always in a game state out of a set S of possible game states. Then playing the game from the point of view of the player is producing an action sequence

$$a_1, ..., a_m;\ a_i \in Act$$

and this action sequence results from the point of view of the game in a sequence

$$s_1, ..., s_m$$

of game states, i.e. $s_i \in S$. If we assume that the two sequences represent a complete play (or run) of the game, then s_m must be a final state of the game and usually we expect that the player has either won the game or lost it, which can be indicated by two predicates $\mathcal{G}_{win}$ and $\mathcal{G}_{lose}$, defined on game states, that are true if the player won the game, resp. lost it. Naturally, many games allow the player to store some states in the sequence $s_1, ..., s_m$ and start the game anew not from the beginning, but from a stored state, let's say s_{save}, but every restart from s_{save} will ultimately lead to a winning or loosing state, again.

While the sets Act and S might be rather large (even seemingly infinite), S is finite (although perhaps not completely known to the designer and programmer) and the set of elements of Act that have a real influence on the successor situation of a situation s is also finite. We define $Act(s) \subseteq Act$ as the set of actions that are enabled (resp. recognized) by the game in situation s. Since $Act(s)$ is finite for each $s \in S$, we can order the actions within $Act(s)$ and can refer to an action by its index within this ordered set. If we want to generate an action sequence randomly, we can generate a sequence of indices ind$_1$,...,ind$_m$ and depending on the game state s_i that was produced by the action sequence a_1,...,a_i so far, the index ind$_{i+1}$ will select the action with this index out of $Act(s_i)$. If $|Act(s_i)| <$ ind$_{i+1}$, then we can just use the index generated by ind$_{i+1}$ modulo $|Act(s_i)|$. In the following, for convenience we will identify actions with indices and vice versa.

A big problem when testing games is the fact that most games determine the successor state s' of a state s not solely based on s and the action $a \in Act(s)$ that the player takes. Even in single-player games, some random effects have some influence on what the next game state will be. For example, in role-playing games, when a character of the human player scores a hit on another character (human or NPC), the damage will not always be the same, it will be computed using a formula that takes the state into account (for example, with what the hit was performed, what protection was worn, etc.), but it also involves the usage of a random number generator. In a sports game, for example simulating soccer, a kick aimed at passing the ball to a certain location near a player of the own team might end up in a slightly different spot that is nearer to a player of the opponent and therefore the human player's team loses the ball. Again, the formula determining the outcome will be based on attributes of the state, for example the skill of the kicking player or the wind conditions, but will also involve some random element.

So, if we execute the action sequence a_1,...,a_m k times, then we end up with k state sequences s_1^1,...,s_m^1;...;s_1^k,...,s_m^k that usually are not identical. Naturally, we also do not want them being totally different. What kind of variety should be observed is one important decision the game designer has to make and usually these decisions are documented as some general behavior rules. So, testing for the achievement of the intended variety is not very easy, since obviously there are a lot of possible action sequences (even of only a certain length) and what we would like to find are action sequences that lead to state sequences that contradict these rules (at least most of the time). So, essentially we have to consider the behavior of a game as something between an indeterministic and a deterministic state automaton where the indeterminism is supposed to obey certain rules.

Another problem regarding testing commercial computer games is the length of action sequences that lead to a winning game state. Totally solving a quest-based game, performing a full race or playing a team sports game for the usually allotted time requires tens of thousands of actions or even more. Fortunately, human players also have problems with this and, in addition to the possibility of saving game states, structuring games into subgoals and consequently into a collection of smaller action sequences each of which solves such a subgoal is a common feature of all commercial computer games. And naturally, testing will make use of this. But the problem of finding sequences revealing unwanted or incorrect behavior still remains. In the next section, we will present an evolutionary approach to tackling this problem.

III. Evolutionary behavior testing

Our approach to testing commercial computer games is aimed at evolving user behaviors that result in game states or game state sequences that are not intended by the game designers. An evolutionary approach to finding such behaviors seems to be the only feasible solution, since conventional search techniques relying on constructing trees or graphs representing all possible behaviors cannot deal with the indeterminism that is part of the game. In addition, the usual control techniques for such conventional techniques rely heavily on good predictions of the consequences of actions, which is nearly impossible for the games and, together with the huge branching factors, this lets the conventional search techniques degenerate to simple enumerations of possibilities. Outside of the use of AI search techniques, there is also not much help to be found. Covering branches in the flow charts representing the game suffers from the sam eproblems as search and does not reflect the high-level expectations of the game designer (although naturally we do expect that such testing, at least for units, is performed).

Using the view on games described in the last section, we propose to use a Genetic Algorithm working on action sequences as individuals to create action sequences that produce an unwanted behavior specified by the fitness function in more than a certain percentage of runs of the game using the action sequence as user interaction input. Obviously, defining the fitness function is the crucial part of this idea. While in the end the concrete fitness function will depend on the game and the unwanted behavior we want to test for, there are some general recommendations, based on the work presented in [2], how such fitness functions can be developed and what they should evaluate.

Formally, our GA is working on the set $\mathcal{F}$ of sequences of indexes for actions, i.e.

$$\mathcal{F} = \{(a_1,...,a_m) \mid a_i \in \{1,..., \max\{|\mathcal{A}ct(s_i)|\}\}$$

The evaluation of the fitness of an individual is based on k runs of this individual as input to the game from a given start game state s_0 (k is a parameter chosen by the tester of the game). Our following definitions are based on associating a low fitness-value with a good individual and a high value with a bad individual. And a good individual is supposed to lead us to unwanted game behavior.

A single run of an individual (a_1,...,a_m) produces a state sequence (s_0,...,s_m) and we define the single run fitness $single_fit$ of (s_0,...,s_m) as follows:

$$single_fit((s_1,...,s_m)) = \begin{cases} j, \\ \quad \text{if } \mathcal{G}_{unwanted}((s_0,...,s_j)) = \text{true} \\ \quad \text{and } \mathcal{G}_{unwanted}((s_0,...,s_i)) \\ \quad = \text{false for all } i < j \\ \sum_{i=1}^{m} near_goal((s_0,...,s_i)), \\ \quad \text{else.} \end{cases}$$

In the formula above, $\mathcal{G}_{unwanted}$ is a predicate on state sequences that decides whether a particular sequence shows an unwanted behavior or not. We expect that there are many different kinds of unwanted behavior and each of these kinds will require their own predicate and subsequently their own evolutionary tests. If we test for unwanted behavior concerned with winning the game, this particular $\mathcal{G}_{unwanted}$ will make use of $\mathcal{G}_{win}$ by requiring that $\mathcal{G}_{win}(s_j)$ = true. But unwanted behavior might also be concerned with the way a human player loses and then an additional condition might be that $\mathcal{G}_{lose}(s_j)$ = true.

A predicate $\mathcal{G}_{unwanted}$ also does not always require to be defined on a whole sequence, it might be that a particular state is unwanted and then we have to test s_j only (for example, programmers might have error states that are for testing purposes only and then finding action sequences that bring the game into such a state is an obvious test). By using as fitness of a run fulfilling $\mathcal{G}_{unwanted}$ the number of actions needed to get to the first game state fulfilling $\mathcal{G}_{unwanted}$, we try to evolve short sequences revealing the unwanted behavior to make analyzing the reasons for the behavior easier.

The key knowledge required for our method is represented by the function $near_goal$ that has to evaluate state sequences that do not fulfill $\mathcal{G}_{unwanted}$ with regard to how near they came to fulfilling $\mathcal{G}_{unwanted}$. As already stated, $near_goal$ will be rather different for different unwanted behaviors and the particular game will also have a big influence on how a particular $near_goal$ has to be defined. If $\mathcal{G}_{unwanted}$ checks for a sequence of game states s_1^{tar},...,s_l^{tar} being part of the sequence to check, then $near_goal$ might evaluate how many of the s_i^{tar} appear in the sequence it is given. If $\mathcal{G}_{unwanted}$ looks for a game state fulfilling certain features, resp. having certain feature values, then $near_goal$ should measure these feature values in the state it is given and sum up the differences (perhaps weighted by some importance measure for features). Essentially, all the experience amassed by evolutionary computing with regard to developing fitness functions will prove useful here.

$single_fit$ sums up the evaluations by $near_goal$ for all subsequences of s_0,...,s_m starting with the subsequence s_0,s_1. On the one hand, this assures in most cases that a state sequence not fulfilling $\mathcal{G}_{unwanted}$ has a $single_fit$-value much higher (usually magnitudes higher) than a sequence that does fulfill $\mathcal{G}_{unwanted}$. On the other hand, we award sequences that come near to fulfilling $\mathcal{G}_{unwanted}$ and stay near to fulfilling it. And finally, we are able to identify actions (resp. indexes in the individual) that move us away from the behavior or result we are looking for (after coming near to it). This will allow us to introduce additional Genetic Operators that proved to be very useful in our experimental evaluation.

With $single_fit$, we are now able to define the fitness of an individual (a_1,...,a_m). As stated before, let (s_0,s_1^1,...,s_m^1),..., (s_0,s_1^k,...,s_m^k) be the state sequences produced by the k evaluation runs for the individual. Then the fitness fit is defined as

$$fit((a_1,...,a_m)) = \sum_{i=1}^{k} single_fit((s_1^i,...,s_m^i))$$

If $near_goal$ leads to $single_fit$-values that are much larger than m, then our definition of fit guarantees that the evolutionary search moves towards action sequences that often lead to fulfilling $\mathcal{G}_{unwanted}$, exactly as we intended. We usually store with an individual not only the fit-value, but also how

many of the runs led to state sequences fulfilling $\mathcal{G}_{unwanted}$ and we report to the tester all action sequences that had more than a given number of such state sequences. It also has proven useful to do a few re-runs for each of the reported sequences to filter out any that were helped in their performance by localized behavior of the game's random number generator.

By using action sequences of a certain length m as individuals, we can use the rather standard Genetic Operators Crossover and Mutation on strings for our evolutionary testing system. As usual, given two action sequences $(a_1,...,a_m)$ and $(b_1,...,b_m)$, Crossover selects randomly a number i between 1 and $m-1$ and produces $(a_1,...,a_i,b_{i+1},...,b_m)$ as new individual. Mutation also selects randomly an i and a random action a out of $Act-\{a_i\}$ to produce $(a_1,...,a_{i-1},a,a_{i+1},...,a_m)$ as new individual.

But we also have a variant of each operator that tries to put some more intelligence into the operators by targeting specific positions in the sequences that are identified by the $near_goal$ evaluations for the fitness. More precisely, when evaluating a game run of an action sequence $(a_1,...,a_m)$, we report the first position a_j such that

$$near_goal((s_0,...,s_j)) \geq near_goal((s_0,...,s_{j-1})) + lose_goal,$$

for a parameter $lose_goal$ that is chosen to indicate a major upset in coming towards fulfilling $\mathcal{G}_{unwanted}$. Then a *Targeted Crossover* between $(a_1,...,a_m)$ and $(b_1,...,b_m)$ first selects one of the reported positions in $(a_1,...,a_m)$, say a_j, then selects a position in $(a_1,...,a_m)$ between $a_{j-COtarg}$ and a_{j-1} and performs a Crossover at this selected position. The same modification leads us to a *Targeted Mutation* using $Mtarg$ to define the interval to choose from.

While Targeted Genetic Operators add another few parameters to a GA (we need in addition parameters determining the relation between targeted and standard operators) and enhance the risk of premature convergence at some local optimum, they add to the evolutionary search process some of the knowledge contained in $near_goal$ by pointing out where it is most likely to improve on the $near_goal$-value of an individual. And by targeting those most likely positions, there is a good chance that some improvement is realized, if such an improvement is possible at all. In this regard, targeted operators are similar to the accountability criteria added to the method of [2] in [1]. It should be noted that early experiments with our system without the use of targeted operators were not very successful. And the comparison done in [1] also suggests that the targeted operators are an important component, but more analysis will be necessary.

IV. CASE STUDY: FIFA-99

In this section, we present some results of using our general method from the last section in testing the game FIFA-99 by Electronic Arts. After a short description of FIFA-99, we present the instantiation of evolutionary behavior testing we used for obtaining the results that we then describe in the third subsection.

A. FIFA-99

FIFA-99 is a typical example of a team sports games. The team sport played in FIFA-99 (in fact, in all games of the FIFA series) is soccer. So, two teams of eleven soccer players square off against each other and the players of each team try to kick a ball into the opponent team's goal (*scoring a goal*). In the computer game, the human player is assigned one of the teams and at each point in time controls one of the team's players (if the team has the ball, usually the human player controls the player that has the ball). In addition to actions that move a player around, the human user can let a player also handle the ball in different ways, including various ways of passing the ball and shooting it. On the defensive side the actions include various tackles and even different kinds of fouls.

The goal of the human user/player is to score more goals than the opponent during the game. FIFA-99 allows the human player to choose his/her opponent from a wide variety of teams that try to employ different tactics in playing the game. In addition, Electronic Arts has included into FIFA-99 an AI programming interface allowing outside programs to control NPC soccer players. For our experiments, we extended this interface to allow for our evolutionary testing system to access various additional information and to feed the action sequences to the game.

B. Instantiating behavior testing for FIFA-99

Naturally, there are many instantiations of our general concept for evolving unwanted behavior within FIFA-99 from Section III. While FIFA-99 is a rather old example of the FIFA series of games, we still do not want to reveal too many behaviors that are considered unwanted by the designers. To nevertheless show the potential of our concept, we present some found action sequences that consistently score goals against selected opponents in a manner that might be acceptable if happening once a game, but that can be repeated nearly all of the time. While we and the game designers do not consider these behaviors as big problems, they can reduce the gaming experience for very good players and allow us to provide one instantiation of the general concept that gives hints how instantiations for searching for different unwanted behaviors can be designed.

A human player of FIFA-99 (and other team sport games) essentially controls one soccer player at each point in time, allowing this player to move, pass and shoot and there is also the possibility for the human player to switch his/her control to another of his/her players. But this switch is also performed automatically by the game if another soccer player of the human player is nearer to the ball (and switch goes through a predefined sequence of the players in the team to accommodate the way the controls for the human player work). So, the set Act for our experiments consisted of

- NOOP
- PASS
- SHOOT
- SWITCH
- UP

- DOWN
- RIGHT
- LEFT
- MOVEUPLEFT
- MOVEUPRIGHT
- MOVEDOWNLEFT
- MOVEDOWNRIGHT

There are some additional actions that are associated with special situations in the game (like THROWIN or CORNER), but for our examples of the next subsection we defined as unwanted behavior any sequence of actions leading from the kickoff to scoring a goal, without having the ball going out or coming into the possession of the other team. So, $\mathcal{G}_{unwanted}((s_0,...,s_i))$ = true, if FIFA-99 reports a goal scored in one of the states $s_o,...,s_i$.

For the function $near_goal$, we divided the playing field into four zones:

Zone 1: from the opponent goal to the penalty box
Zone 2: 1/3 of the length of the field from the opponent goal
Zone 3: the half of the field with the opponent goal in it
Zone 4: the whole field.

We associate with each zone a penalty value (pen$_1$ to pen$_4$) and decide which value to apply based on the position of the player with the ball (resp. if none of the own players has the ball, then we look at the position of the player that had the ball in the last state). If the position is in zone 1, we apply pen$_1$ as penalty, if the position is not in zone 1, but in zone 2, we apply pen$_2$ and so on. By $dist(s_i)$ we denote the distance of the player position from above to the opponent's goal. Then for a state sequence (s_0,...,s_i), we define $near_goal$ as follows:

$$near_goal((s_0, ..., s_i)) = \begin{cases} dist(s_i) \times \text{penalty}, \\ \quad \text{if the own players had the ball} \\ \quad \text{in } s_{i-1} \text{ or } s_i \\ max_penalty \\ \quad \text{else.} \end{cases}$$

The parameter $max_penalty$ is chosen to be greater than the maximal possible distance of a player with the ball from the opponent's goal multiplied by the maximal zone penalty, so that losing the ball to the opponent results in large $near_goal$-values and a very bad fitness. The penalties for the zones allow us to either drive the evolutionary search towards getting near the opponent goal quickly (if they are the same value or higher with higher zone number) or to stay away from the goal for some time (if the lower zone numbers get slightly higher penalty values; this is what we used in our experiments to draw the opponent off guard).

For our Targeted Genetic Operators, we choose $lose_goal$ as $max_penalty$ and did not consider $near_goal((s_0,...,s_{j-1}))$ at all, so that we targeted those actions in a sequence that resulted in a loss of the ball. We used $Mtarg$ = $COtarg$ = 1, so that we had a clearly defined position (and not an interval) and we always selected the smallest position that resulted in a loss of the ball in any of the evaluation runs. We selected Targeted Operators 4 times more often than the standard versions.

Fig. 1. Kickoff

C. Some experimental findings

We used the instantiation of our general method described in the last subsection to run several test series. Initially, we flagged every action sequence produced in a series that fulfilled $\mathcal{G}_{unwanted}$ in each evaluation run (with k=10) and every series had at least one of them. But some of these flagged sequences did not score goals, when we repeated the runs (with us observing them at normal speed), so we added an afterwards automatic filtering of the flagged sequences by running them 100 times and only letting those sequences through that resulted at least 80 times in goals (compared to the number of individuals we generated over all the test series, these additional evaluation runs are neglectable). It should be noted that we also did not do all 100 runs one immediately after the other, but delayed them a little bit, since it became obvious that the random number generator used in FIFA-99 was not so random when called very often in short time intervals. We still have around 100 sequences that passed this automatic filter.

In the following, we present first three sequences produced in the same series that show the effects of the evolutionary learning our method performs very nicely. And then we present a sequence that reveals a really unwanted behavior (remember, being able to score is not really unwanted, we use it here to show what is possible). Fig. 1 shows the start state s_0, the kickoff. In all figures, the light-colored soccer players are the ones controlled by the action sequences and the dark-colored players are the opponents. The player currently executing the actions given by the sequence is indicated by the square target displayed below it. The white figure standing below the center point in Fig. 1 is the referee (it will not appear in any other figure). If you have the pdf-file of this paper available, you can zoom into the figures and you will be able to see additional information about movement vectors, but for understanding the observed behaviors the greyscale figures and our explanantions of what is happening should be enough.

Fig. 2 shows the last game state that the three action

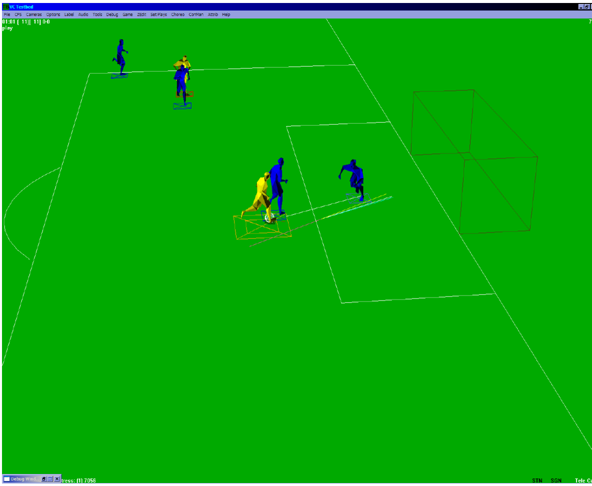

Fig. 2. Evolving a play: one defender and goalie to beat

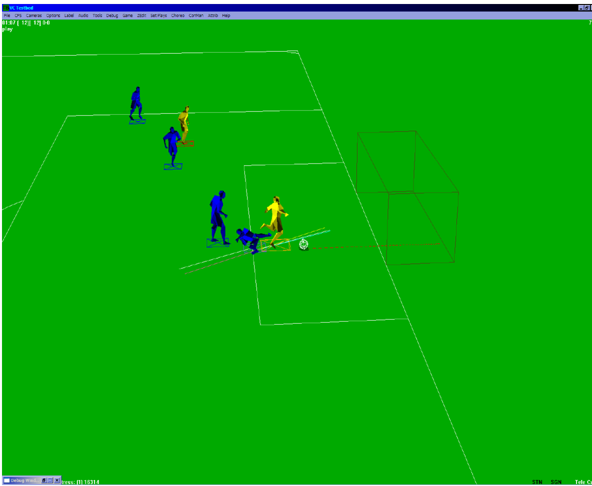

Fig. 3. First sequence: stumble ...

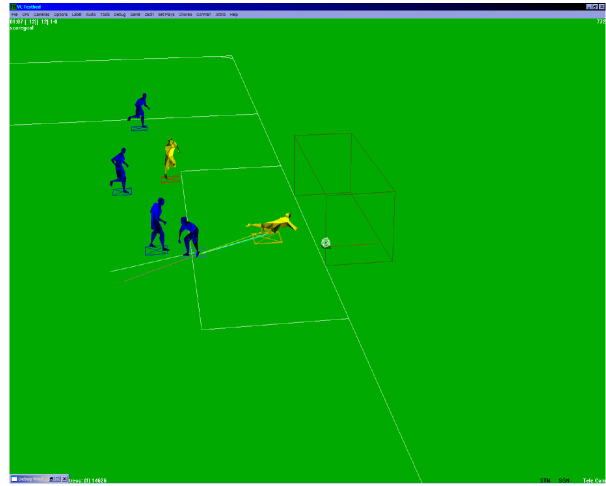

Fig. 4. First sequence: ... and score

Fig. 5. Second sequence: deceive ...

sequences produced by one test run have in common. Our player in the middle of the picture is passed the ball and gets it under control with one defender and the goalie to beat (Fig. 2).

The first sequence evolved by our system sees our player dribbling around very near the goalie (Fig. 3) and while the player is fouled by the goalie (Fig. 4), the ball has enough forward motion to get into the goal. While this is a behavior that can be observed in real soccer, remember that it can be repeated many times, which is not realistic.

Out of the first scoring sequence, a second one was evolved in which our player makes a feint, deceives the goalie so that it cannot foul (Fig. 5) and then walks into the goal with the ball (Fig. 6). Given the fact that there is still a defender near, the walking into the goal is a little risky, but our system did evolve a third sequence that again has a deceived goalie (Fig. 7) and then our player kicks the ball into the empty net (Fig. 8), leaving the defender no chance.

The Figures 9 to 14 show the final stages of another successful and interesting sequence. After passing the ball around several times before and slightly behind midfield, a winger gets the ball and, as shown in Fig. 9, this winger kicks (or passes) the ball towards the goal line near the corner. The winger and a defender race to the ball (Fig. 10) and it looks like the ball will go out (although it slows down considerably, Fig. 11). The defender definitely thinks that it will go out and slows down, so that our attacker can get to defender and ball (Fig. 12) and just before the ball would have gone out, our attacker centers the ball (Fig. 13) to a free player near the short corner of the goal and this free player scores (Fig. 14). Again, once in a game this would be a rather realistic (and amusing) play, but the fact that it can be repeated over and over again without the defender learning from the mistake is an unwanted behavior.

All the sequences that our testing system found play around the midfield for some time, before a deep pass occurs (in different sequences to different players in different parts of the field). Many of the sequences have then some short pass play and dribbling going on, before a player rather near to the goal has a good opportunity to shoot and consequently scores.

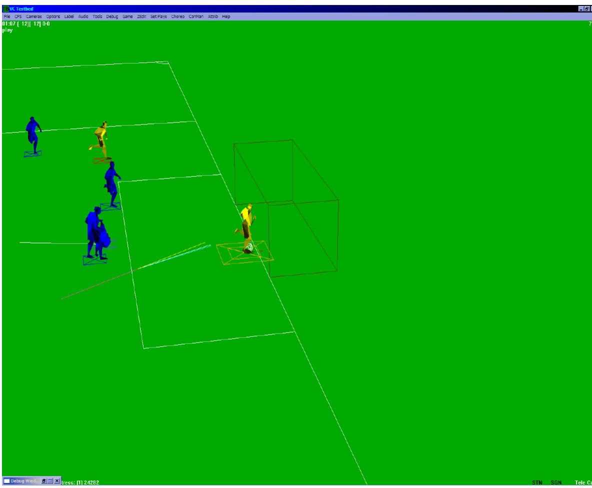

Fig. 6. Second sequence: ... and walk in

Fig. 8. Third sequence: ... and tip it in

Fig. 7. Third sequence: deceive ...

Fig. 9. Passing strongly to goal line

The sequences are rather long and therefore the opponent has lots of opportunities to break up the plays, but this happens only very seldom (having probabilities of intercepting the ball rise would be a remedy to get rid of the problem that these plays work so often). It should also be noted that for a human player it is rather difficult to repeat the action sequences with good enough timing to make use of the revealed weaknesses.

V. Conclusion and Future Work

We presented a general testing method for commercial computer games based on evolving user action sequences producing unwanted (or to be tested) behavior. The strength of evolutionary methods, the combination of intuition-like guidance and exploitation of knowledge, allows our method to produce user behavior that leads to fulfilling rather high-level testing goals, something that so far has been very difficult to achieve even for human testers. Exploiting this general method should allow the games industry to keep track with the growing complexity of their games on the testing side, but we also hope that interesting evolved behavior will find its way back into the game AI (in a way controlled by the game designer). Naturally, defining appropriate fitness functions is still the key problem, but the games themselves and their state representations offer very good candidates for measures already.

Our current implementation of our method for the game FIFA-99 is only a first step. FIFA-99 already has some special actions that require from a human player to enter parameters for the actions (in a graphical manner). Dealing with such actions will require from us to evolve appropriate parameter values for them, which needs additional Genetic Operators on the parameters. Actions with parameters would also allow to better treat the human reaction time that influences the time intervals between non-NOOP actions. For team sport games like FIFA, the length of action sequences of interest is not large. Role-playing games will require longer sequences, which might require switching from the off-line learning approach we are using now to an on-line approach (perhaps making use of the techniques developed in [3]).

Fig. 10. A race?

Fig. 11. No danger?

Fig. 12. Why did you slow down?

Fig. 13. This center should never have happened!

Fig. 14. Goal!!!

ACKNOWLEDGEMENTS

This work was supported by IRIS within the IACCG project.

REFERENCES

[1] J. Denzinger and S. Ennis. Improving Evolutionary Learning of Cooperative Behavior by Including Accountability of Strategy Components, Proc. MATES 2003, Erfurt, Springer LNAI 2831, 2003, pp. 205–216.

[2] J. Denzinger and M. Fuchs. Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, Kyoto, 1996, pp. 48–55.

[3] J. Denzinger and M. Kordt. Evolutionary On-Line Learning of Cooperative Behavior with Situation-Action Pairs, Proc. ICMAS'00, IEEE Press, 2000, pp. 103–110.

[4] D.B. Fogel. Blondie24: Playing at the Edge of AI, Morgan Kaufmann, 2002.

[5] J.E. Laird and M. van Lent. Human-level AI's killer application: Interactive computer games, AI Magazine, 22(2), 2001, pp. 15–25.

[6] J. Schaeffer. One Jump Ahead – Challenging Human Supremacy in Checkers, Springer, 1996.